AFRL-SN-WP-TR-2002-1103

SELECTION, COMBINATION, AND EVALUATION OF EFFECTIVE SOFTWARE SENSORS FOR DETECTING ABNORMAL USAGE OF COMPUTERS RUNNING WINDOWS NT/2000

Jude Shavlik, Ph.D.

Shavlik Technologies, LLC
4750 White Bear Parkway
White Bear Lake, MN 55110

APRIL 2002

Final Report for 12 October 2000 – 10 April 2002

SENSORS DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7318

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YY)*** April 2002	**2. REPORT TYPE** Final	**3. DATES COVERED** ***(From - To)*** 10/12/2000 – 04/10/2002

4. TITLE AND SUBTITLE SELECTION, COMBINATION, AND EVALUATION OF EFFECTIVE SOFTWARE SENSORS FOR DETECTING ABNORMAL USAGE OF COMPUTERS RUNNING WINDOWS NT/2000	**5a. CONTRACT NUMBER** F33615-00-C-1745
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 69199F
6. AUTHOR(S) Jude Shavlik, Ph.D.	**5d. PROJECT NUMBER** ARPS
	5e. TASK NUMBER NZ
	5f. WORK UNIT NUMBER 08
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Shavlik Technologies, LLC 4750 White Bear Parkway White Bear Lake, MN 55110	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Sensors Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson Air Force Base, OH 45433-7318	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/SNZW
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-SN-WP-TR-2002-1103

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Intrusion-detection systems (IDS) can either (a) look for known attack patterns, or (b) be adaptive software that is smart enough to monitor and learn how the system is supposed to work under normal operation versus how it works when misuse is occurring. They used approach (b) in this project. Specifically, they empirically determined which sets of fine-grained system measurements are the most effective at distinguishing usage by the assigned user of a given computer from misusage by other insiders within an organization. In this project, they have made significant advances toward creating an IDS that requires few CPU cycles (less than 1 percent), produces few false alarms (less than one per day), and detects most intrusions quickly (about 95 percent within 5 minutes). The algorithm that was developed measures over 200 Windows 2000 properties every second, and creates about 1500 features out of them. During a machine-learning training phase, the algorithm learns how to weight these 1500 features in order to accurately characterize the particular behavior of each user–each user gets his or her own set of feature weights. Following training, every second all of the features vote as to whether or not it seems like an intrusion is occurring. The weighted votes for and against an intrusion are compared, and if there is enough evidence, an alarm is raised.

15. SUBJECT TERMS
user profiling, access control, intrusion detection, learning systems, intrusion detection systems, user identification via usage profiling

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	SAR	46	Martin R. Stytz, Ph.D. **19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-2811 x4380

HES&S 31-15093-1

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

1. Introduction

In an increasingly computerized and networked world, it is crucial to develop defenses against malicious insider activity in information systems. One promising approach is to develop computer algorithms that detect insiders who are inappropriately intruding on the computers of others. However, *intrusion detection* is a difficult problem to solve [DARPA99]. System performance cannot be adversely affected, false positives must be minimized, and intrusions must be caught (i.e., false negatives must be very low). The current state of the art in intrusion-detection systems is not good; false positives are much too high and successful detection is unfortunately too rare. In this project we have made significant advances toward creating an intrusion-detection system that requires few CPU cycles (less than 1%), produces few false alarms (less than one per day), and detects most intrusions quickly (about 95% within five minutes).

Intrusion-detection systems (IDS's) can either (a) look for known attack patterns or (b) be "adaptive software" that is smart enough to monitor and learn how the system is supposed to work under normal operation versus how it works when misuse is occurring [LUNT93]. We address approach (b) in this project. Specifically, we are empirically determining which sets of fine-grained system measurements are the most effective at distinguishing usage by the assigned user of a given computer from misusage by other "insiders" [DARPA99; NEUMANN99] within an organization.

Building on our expertise in Windows 2000 computer security and machine learning, we have written and evaluated a prototype anomaly-detection system that creates statistical profiles of the normal usage for a given computer running Windows 2000. Significant deviations from normal behavior indicate that an intrusion is likely occurring. For example, if the probability that a specific computer receives 10 Mbytes/sec during evenings is measured to be very low, then when our monitoring program detects such a high transfer rate during evening hours, it can suggest that an intrusion may be occurring.

The algorithm we have developed measures over two-hundred Windows 2000 properties every second, and creates about 1500 "features" out of them. During a machine-learning "training" phase, it learns how to weight these 1500 features in order to accurately characterize the particular behavior of each user – each user gets his or her own set of feature weights. Following training, every second all of the features "vote" as to whether or not it seems like an intrusion is occurring. The weighted votes "for" and "against" an intrusion are compared, and if there is enough evidence, an alarm is raised. (Section 2 presents additional details about our IDS algorithm that are being glossed over at this point.)

This ability to create statistical models of individual computer's normal usage means that each computer's unique characteristics serve a protective role. Similar to how each person's antibodies can distinguish one's own cells from invading organisms, these statistical-profile programs can, as they gather data during the normal operation of a computer, learn to distinguish "self" behavior from "foreign" behavior. For instance, some people use Notepad to view small text files, while others prefer WordPad. Should someone leave their computer unattended and someone else try to inappropriately access their files, the individual differences between people's computer usage will mean that our statistical-modeling program will quickly recognize this illegal access.

We evaluate the ability to detect insider misuse by collecting data from multiple employees of Shavlik Technologies, creating user profiles by analyzing "training" subsets of this data, and then experimentally judging the accuracy of our approach by predicting whether or not data in "testing sets" is from the normal user of a given computer or from an intruder. The key scientific hypothesis investigated is whether or not creating statistical models of user behavior can be used to accurately detect insider abuse. We focus our algorithmic development on methods that produce very low false-alarm rates, since a major reason system administrators ignore IDS systems is that they produce too many false alarms.

Our empirical results suggest that it is possible to detect about 95% of the intrusions with less than one false alarm per (8 hr) day per user. It should be noted, though, that these results are based on our model of an "insider intruder," which assumes that when Insider *Y* uses User *X*'s computer, that *Y* is not able to alter his or her behavior to explicitly mimic *X*'s normal behavior. The training phase our approach can be computationally intensive due to some parameter tuning, but this parameter tuning could be done on a central server or during the evenings when users are not at work. The CPU load of our IDS is negligible during ordinary operation; it requires less than 1% of the CPU cycles of a standard personal computer. Our approach is also robust to the fact that users' normal behavior constantly changes over time.

Our approach can also be used to detect abnormal behavior in computers operating as specialized HTTP, FTP, or email servers. Similarly, these techniques can be used to monitor the behavior of autonomous intelligent software agents in order to detect rogue agents whose behavior is not consistent with the normal range of agent behavior for a given family of tasks. However, the experiments reported herein only involve computers used by humans doing the normal everyday business tasks. While we employ the word "user" throughout this report, the reader should keep in mind that, except for the technique and experiments involving typing patterns (Section 4), our approach applies equally well to the monitoring of servers and autonomous intelligent agents. All that would be needed to apply our approach to a different scenario would be to define a set of potentially distinctive properties to measure and to write code that measured these properties periodically.

Previous empirical studies have investigated the value of creating intrusion-detection systems by monitoring properties of computer systems, an idea that goes back at least 20 years [ANDERSON80]. However, prior work has focused on Unix systems, whereas over 90% of the world's computers run some variant of Microsoft Windows. In addition, prior studies have not looked at as large a collection of system measurements as we use. For example, Warrender et al. [WARRENDER99], Ghosh et al. [GOSH99], and Lane and Brodley

[LANE98] only look at Unix system calls, whereas Lee et al. [LEE99] only look at audit data, mainly from the TCP program.

In Section 2 we describe the algorithm we developed that analyzes the Windows 2000 properties that we measure each second, creating a profile of normal usage for each user. Section 3 presents and discusses empirical studies that evaluate the strengths and weaknesses of this algorithm, stressing it along various dimensions such as the amount of data used for training. This section also lists which Windows 2000 properties end up with the highest weights in our weighted-voting scheme. Section 4 reports some additional experiments we performed that monitor keystroke data in order to identify users, which we can also do accurately while generating few false alarms. Section 5 describes possible future follow-up research tasks, and Section 6 concludes this final project report.

2. Algorithm Developed

In this section we describe the primary algorithm that we developed in the course of this project. Arguably our key finding is that a machine-learning algorithm called Winnow [LITTLESTON88], a weighted-majority type of algorithm, works very well as the core component of our IDS.

This algorithm operates by taking weighted votes from a pool of individual prediction methods, continually adjusting these weights in order to improve accuracy. In our case, the individual predictors are the Windows 2000 properties that we measure, where we look at the probability of obtaining the current value and comparing it to a threshold. That is, each measurement suggests that an intrusion may be occurring if:

$$\text{Prob (measured property has value } v) \; < \; p \qquad \text{[Eq. 1]}$$

Each property we measure votes as to whether or not an intrusion is currently occurring. When the weighted sum of votes leads to the wrong prediction (intrusion vs. no intrusion), then the weights of all those properties that voted incorrectly are halved. Exponentially quickly, those properties that are not informative end up with very small weights. Besides leading to a highly accurate IDS, the Winnow algorithm allows us to see which Windows 2000 properties are the most useful for intrusion detection, namely those properties with the highest weights following training (as we shall see, when viewed across several users, a surprisingly high number of properties end up with high weights).

Before presenting our algorithm that calls as subroutine the Winnow algorithm, we discuss how we make "features" out of the over two-hundred Windows 2000 properties that we measure. Technically, it is these 1500 or so features that do the weighted voting.

2.1 Features Used

Appendix A lists and briefly describes all of the Windows 2000 properties that we measure; some relate to network activity, others to file accesses, some to the CPU load, and others to the identity of the programs currently running. For each of the measurements described in Appendix A, we also derive several additional measurements:

```
Actual Value Measured
Average of the Previous  10 Values
Average of the Previous 100 Values
Difference between Current Value and Previous Value
Difference between Current Value and Average of Last  10
Difference between Current Value and Average of Last 100
Difference between Averages of Previous 10 and Previous 100
```

As will be seen in our experiments, these additional "derived" features play an important role; without them intrusion-detection rates are significantly lower. For the remainder of this report, we will use the term "feature" to refer the combination of a measured Windows 2000 property and one of the seven above transformations. In other words, each Windows 2000 property that we measure produces seven features. (The first item in the above list is not actually a *derived* feature; it is the "raw" measurement but we include it in the above list for completeness.)

For one of the experiments described below, we also use four more derived measurements:

```
Average of the Previous 1000 Values
Difference between Current Value and Average of Last 1000
Difference between Averages of Previous 10 and Previous 1000
Difference between Averages of Previous 100 and Previous 1000
```

2.2 Our IDS Algorithm

Table 1 contains the algorithm that is the primary contribution of this project. We take a machine-learning [MITCHELL97] approach to creating an IDS, and as is typical we divide the learning process into three phases. First, we use some *training data* to create a model; here is where we make use of the Winnow algorithm (see Table 2). Next, we use some more data, called the *tuning set*, to "tune" some additional parameters in our IDS. Finally, we evaluate how well our learned IDS works be measuring its performance on some *testing data*. We repeat this process for multiple users and report the average *test-set* performance in our experiments.

The Windows 2000 properties that we measure are continuous-valued, and in Step 1b of Table 1 we first decide how to discretize each measurement into 10 bins; we then use these bins to create a discrete probability distribution for the values for this feature. Importantly, we do this discretization separately for each user, since this way we can accurately approximate each user's probability distribution with our 10 bins. (We did not experiment with values other than 10 for the number of bins.)

We always place the value 0.0 in its own bin, since it occurs so frequently. We then choose the "cut points" that define the remaining bins by fitting a sample of the measurements produced by each user to each of several standard probability distributions: uniform, Gaussian, and Erlang (for k ranging from 1 to 100). When $k = 1$ the Erlang is equivalent to the better known ("decaying") Exponential distribution, and as k increases the distribution looks more and more like a Gaussian. We then select the probability distribution that best fits the sample data, and create our 10 bins as follows:

For the *uniform* probability distribution, we uniformly divide the interval [minimum value, maximum value] into seven bins, and use the two remaining bins for values less than the minimum and greater than the maximum (since values encountered in the future might exceed those we have seen so far).

For the *Gaussian* probability distribution, we place the lowest 0.005% of the probability mass in the first bin, the next 0.1% in the second bin, 5% in the next bin, and 15% in the bin after that. We do the same working down from the highest value, which leaves about 60% for the middle bin (60% is roughly one standard deviation around the mean of a Gaussian).

For the *Exponential* probability distribution, we put half the probability mass in the first bin, and then half of the remaining probability mass in each successive bin.

For the *Erlang* probability distribution, we execute a combination of what we do for the Gaussian and the Exponential, depending on the value of k.

Most of our features are best modeled by Gaussians, with the Exponential distribution being the second most common selection. One final point about converting to a discrete probability distribution needs to be mentioned: for those Windows 2000 measurements that vary over orders of magnitude (e. g., bytes sent per second), we use the *log* of their values.

After we have discretized our features, we simply count how often in the training data did a feature value fall into a given bin, thereby producing a probability distribution (after normalizing by the total number of counts). Following standard practice, we initialize all bins with a count of 1; this ensures that we will never estimate from our finite samples a probability of zero for any bin. We are now able to estimate the Prob (feature = measured *value*) that was mentioned earlier in Eq. 1.

We next turn to discuss using these probabilities to learn models for distinguishing the normal user of a given computer from an intruder. Ideally we would use training data where some User *X* provided the examples of normal (i. e., non-intrusion) data and we had another sample of data measured during a wide range of intrusions on this user's computer. However, we do not have such data (this is a problem that plagues IDS research in general), and so we use what is a standard approach, namely we collect data from several users (in our case, 10), and we then simulate intrusions by replaying User *Y's* measurements on User *X*'s computers. We say that a *false alarm* occurs when User

X's recent measurements are viewed as anomalous - that is, suggestive of an intrusion - when replayed on his or her own computer. A *detected intrusion* occurs when we view User *Y*'s measurements as being anomalous when evaluated using *X*'s feature discretization and feature weighting. (Notice that we need to use *X*'s discretization, rather than *Y*'s, since we are assuming that *Y* is operating on *X*'s computer.)

Table 2's version of Littlestone's Winnow algorithm is used to choose weights on the features we measure. This algorithm is quite simple, yet has impressive theoretical properties [LITTLESTONE88] and practical success on real-world tasks, especially those that have a very large number of features, which is the case for our project. As already mentioned, this algorithm sums weighted votes "for" and "against" the possibility that an intrusion is currently occurring. When the winning choice (i. e., "for" or "against") is wrong, then all those features that voted for the wrong choice have their weights halved. We perform the Winnow algorithm for each user, in each case using a 50-50 mixture of examples, with half drawn from this user's measured behavior (the "*against* an intrusion" examples) and half randomly drawn from some other user in the experiment (the "*for* an intrusion" examples).

In order to raise an alarm after the training phase (Step 1 in Table 1) has set the feature weights, our algorithm does not simply use the current weighted vote. Instead, the current weighted vote can raise what we call a *mini alarm*, and we require that there be at least N mini alarms in the last W seconds in order to raise an actual alarm (see Steps 2b and 2c in Table 1). As will be seen in Section 3, W needs to be on the order of 100 to get good detection rates with few false alarms.

We choose the settings for our parameters *on a per-user basis* by evaluating performance on a set of *tuning* data – see Step 2 of Table 1. One significant advantage of a data-driven approach like ours is that we do not have to pre-select parameter values. Instead, the learning algorithm selects for each user his or her personal set of parameter values, based on the performance of these parameters on a substantial sample of "tuning set" data.

The only computationally demanding portion of our algorithm is the parameter-tuning phase, which depends on how many parameter combinations are considered and on how much tuning data each combination is evaluated. In a fielded system, it might make sense to do this step on a central server or during the evenings. The other tasks of measuring features, computing weighted sums, and using Winnow to adjust weights can all be done very rapidly. Outside of the parameter tuning, Table 1's algorithm requires less than 1% of a desktop computer's CPU cycles.

Notice that even during the testing phase (e. g., Step 3 in Table 1), we find it necessary to still execute the Winnow algorithm, to adjust the weights on the features after our algorithm decides whether or not an intrusion occurred. If we do not do this, we get too many false alarms when the user's behavior switches and the intrusion-detection rates drops to 20% from about 95%. On the other hand continually adjusting weights means that if we *miss* an intrusion we will start learning the behavior of the intruder, which is a

Table 1. Algorithm Description: Creating and Maintaining an IDS for User *X*

Step 1: Initial Training

Step 1a: Collect measurements from User *X* for *N* days and place in TRAINSET.

Step 1b: Using TRAINSET, choose good "cut points" (for User X) to discretize continuous values. See text for further explanation.

Step 1c: Select weights for User *X*'s measured features by applying the Winnow algorithm (see Table 2 and accompanying text) using TRAINSET and an equal number of "archived" sample measurements from other users However, be sure to discretize the measurements from the other users by applying User *X*'s cut points, since we will be pretending that the other users are inappropriately using *X*'s computer.

Step 2: Parameter Tuning

Step 2a: While collecting measurements from User *X* for *M* additional days, perform Steps 2b and 2c, calculating *false-alarm* and *intrusion-detection* rates in conceptually independent runs for as many as possible combinations of the parameters being tuned: W, $\text{threshold}_{\text{mini}}$ and $\text{threshold}_{\text{full}}$.

Step 2b: Use the weighted features to "vote" on "mini-alarms" each second; if ($\text{weightedVotes}_{\text{FOR}}$ / $\text{weightedVotes}_{\text{AGAINST}}$) > $\text{threshold}_{\text{mini}}$ then raise a *mini-alarm*. See Steps 2a and 2b of Table 2.

Step 2c: If the number of *mini-alarms* in the last W seconds ≥ $\text{threshold}_{\text{full}}$ then raise an alarm signal that an intrusion might be occurring.

Step 2d: Apply the Winnow algorithm to the data collected during the previous W seconds. This allows the feature-weighting algorithm to track changes in User *X*'s computer usage over time. (Our experiments demonstrated that if we do not continually learn, the system's performance is substantially reduced.)

Notice that our methodology is fair in that we do not perform the learning step until *after* we have made a decision regarding whether or not to sound an alarm. For example, in a fielded system, User *X* might need to reauthenticate him or herself after an alarm is raised, and only after a successful reauthentication should the Winnow algorithm be invoked.

Step 2e: Assuming the desired maximum false-alarm rate is *P* per (8-hour) day, choose the parameter settings that produce the highest intrusion-detection rate on the set of sample "other" users, while not producing more than the desired number of false alarms for User *X*.

Step 3: Continual Operation

Using Step 2e's chosen settings for W, $\text{threshold}_{\text{mini}}$ and $\text{threshold}_{\text{full}}$, repeat Steps 2b through 2d forever. (It might make sense to periodically reselect good parameter settings, say once a month. However, we did not evaluate doing so in the experiments we report herein.)

Table 2. The Winnow Algorithm

```
Step 1: Initialize User X's weights on each feature measured to 1.
Step 2: For each training example do:
    Step 2a: Set weightedVotes_FOR = 0 and weightedVotes_AGAINST = 0.
    Step 2b: If then probability of the current measured value for feature $f < p$,
             then add weight_f to weightedVotes_FOR
             otherwise add weight_f to weightedVotes_AGAINST.

             I.e., if the probability of the current value of feature f is "low,"
             then this is evidence that something anomalous is occurring.
             (In our experiments, we found that $p = 0.15$ was a good setting;
             however, overall performance was robust in regards to the value
             of p. Various values tried between 0.05 and 0.75 all worked well.)
    Step 2c: If weightedVotes_FOR > weightedVotes_AGAINST
             then call the current measurements anomalous.
    Step 2d: If User X produced the current measurements and they were considered
             anomalous, then a false-alarm error has been made.
             Multiply by ½ all those features that incorrectly voted for raising an alarm.

             Otherwise if some other user produced the current measurements and they
             were not considered anomalous, then a missed-intrusion error has been made.
             Multiply by ½ all those features that incorrectly voted against raising an alarm.

             When neither a false-alarm nor a missed-intrusion error occurred, leave the
             current weight unchanged.
```

weakness of our approach (and a weakness of statistics-based approaches for intrusion detection in general). This also means that the empirical results reported in the next section should properly be interpreted as estimating the probability that we will detect an intruder after his or her *first* W seconds of activity. A subject for future work is to empirically evaluate how likely our approach will detect an intruder in the *second* (and successive) W seconds of activity, given we did not detect the intruder in the first W seconds. On the other hand, the fact that we continually are adjusting the weights means that after the legitimate user reauthenticates him or herself after a false alarm, our algorithm will adapt to the change in the user's behavior.

Obviously there is a delicate balance between adapting quickly to changes in the legitimate user's behavior, and thus reducing false alarms, and adapting too quickly to the activity of an intruder and thus thinking the intruder's behavior is simply a change in the behavior of the normal user of the given computer and thereby missing actual intrusions. It is a simple fact of life that most users' behavior is wide ranging and changing over time. The more consistent a user's behavior is, and the more accurately we can capture his or her idiosyncrasies, the better our approach will work.

3. **Experimental Evaluation**

This section reports our experimental evaluation of the algorithm in Table 1. We first describe our experimental methodology, then follow that with experimental results and associated discussion.

When reading the empirical results in this section, it should be remembered that there can be a several percentage points of variation across various experimental runs, due to changes in parameter settings, as well as the sizes and particular contents of the training, tuning, and testing sets used. Differences of a couple of percentage points in detection rates should not be interpreted as significant, but rather as *suggestive*. A large-scale experiment would be needed to reliably estimate, say, 95%-confidence intervals on the detection rates. Also, except in one curve (Figure 1), we do not report the false-alarm rates associated with the intrusion-detection rates. Instead, we only report scenarios where the false-alarm rate is less than one per work day per user, a number we choose as our design specification. Often the actual false-alarm rates on the testing sets are as low as one every 5-7 days, especially for larger values of Table 1's parameter W.

It is also very important to remember that all the results reported in this section are measured on *testing* data that is *not* used during the training and tuning process. In all cases we get *zero* false-alarm rates on the tuning data, as well as higher intrusion-detection rates. It is considered a "fatal flaw" in machine-learning research to report (as representative of future accuracies) the results on data used during the training and parameter-tuning processes, since it is relatively easy to get unrealistically high accuracies when one has the freedom to adjust parameters. What is important is the performance on *future* data, and the results on the testing data provides what is called an "unbiased" (i. e., fair) estimate since these results are obtained after all the parameters in the learning system have been set.

3.1 Methodology

We collected about 8 GB of data from 16 employees of Shavlik Technologies who volunteered to be experimental subjects. Recall that Appendix *A* describes those system properties that we measure once every second. Of these 16 experimental subjects, we use the 10 from whom we have the most data as our pool of "insiders," i.e., they are intended to represent a small department of co-workers. We use these 10 during training (Steps 1 and 2 of Table 1); for each one, we train an IDS to recognize the differences in behavior of that user from the other 9 users. The remaining 6 subjects, for whom we have a total of about 50 work days of measurements, serve as simulated "external" intruders, i.e., users whose computer-usage behavior has not been seen before – these 6 experimental subjects are only used during the *testing* phase (Step 3 of Table 1) and are never used during the training and tuning phases. Hence, one expects that these 6 "outsiders" would be harder to recognize as intruders on User *X*'s computer since their behavior is not observed while the IDS's are still learning.

In all of our experiments we only use data measured between 9am and 5pm on weekdays (i.e., Monday-Friday). For each of the 10 "insiders," we divided their data into disjoint training (Step 1 of Table 1), tuning (Step 2 of Table 1), and testing (Step 3 of Table 1) sets, with at least two weeks of data in each of these three sets. (Also, for each user all the training data was measured earlier than all of the tuning and testing data, and all of the tuning data was measured earlier than all of the testing data. In other words, each subject's three datasets are not temporally intermingled.)

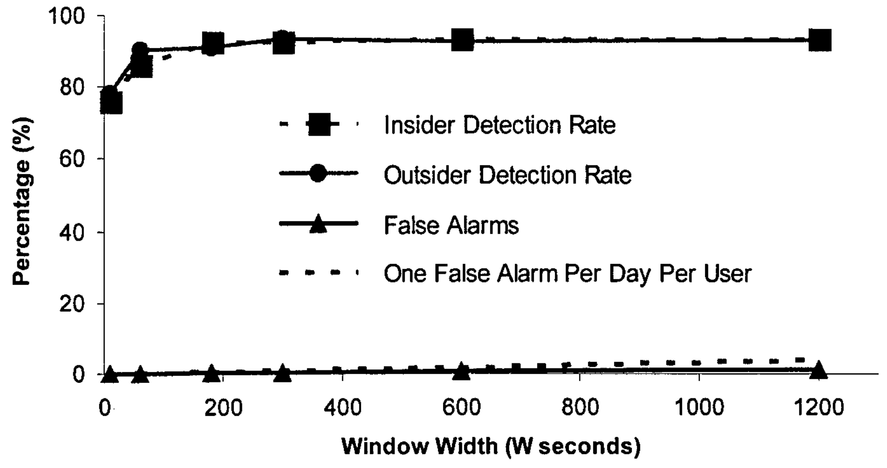
3.2 Basic Results and Discussion

Figure 1 shows, as a function of W (see Table 1) the detection and false-alarm rates for the scenario where the training lasts 15 work days (15 x 8 x 60 x 60 = 432,000 seconds), and the tuning, and testing periods each last 10 work days (288,000 seconds). This scenario involves a five-week-long training process, but as presented in Section 3.3 shorter training periods produce results nearly as good.

The results are averages over the 10 "insiders;" that is, each of these 10 experimental subjects is evaluated using the other 9 subjects as "insider intruders" and the above-described 6 "outsider intruders," and the 10 resulting sets of false-alarm and detection rates are averaged to produce Figure 1. During the tuning phase of Table 2, the specified false-alarm rate of Step 2e was set to 0; such a extreme false-alarm rate could always be produced on the tuning set, though due to the fact we are able to explicitly fit our parameters only to the tuning data, a false-alarm rate of zero did not result during the testing rate (as one expects). For W = 60 sec we are not able to consistently achieve our goal of no more than one false alarm per day per user on the testing data when tuning Table 1's parameter $\text{threshold}_{\text{mini}}$, so we fixed $\text{threshold}_{\text{mini}} = 1$ for W = 60 sec. We further discuss over fitting (getting much higher accuracies on the tuning data than on the testing data due to having too many "degrees of freedom" during the tuning phase), in Section 3.3. Over fitting is arguably the key issue in machine learning.

One potentially confusing technical point needs to be clarified at this point. In an eight-hour work day, there are 480 sixty-second-wide windows (i. e., W = 60) but only 48 six-hundred-second-wide (W = 600) ones. So one false alarm per day for W = 60 sec corresponds to a false-alarm rate of 0.2% whereas for W = 600 sec a false-alarm rate of 2.1% produces one false-alarm per day on average. The (lower) dotted line in Figure 1 shows the false-alarm rate that produces one false alarm per day per user. As can be seen after some thought, as W increases the actual number of false alarms per day decreases. Conversely, as W increases an intruder is able to use someone else's computer longer before being detected.

Figure 1. False Alarm and Detection Rates on Test-Set Data



As can be seen in Figure 1, for a wide range of window widths (from 1 to 20 minutes), the false-alarm rates are very low – always less than one per eight-hour work day per user - and the intrusion-detection rates are impressively high, nearly 95%. Interestingly, the detection rate for "outsiders," whose behavior is never seen during training, is approximately the same as for "insiders." This suggests that our learning algorithm is doing a good job of learning what is characteristic about User *X*, rather than just exploiting idiosyncratic differences between User *X* and the other nine "insiders."

Based on Figure 1, 300 seconds is a reasonable setting for W in a fielded system, and in most of the subsequent experiments in this section use that value.

(It should be noted that going down to W = 60 sec in Figure 1 is not completely appropriate. Some of the features we use are averages of a given measurement over the last 100 seconds, as explained earlier in this report. In all of our experiments, we do not use any examples where the user's computer has not been turned on for at least 100 seconds. Hence, when we replay a 60-second window of activity from User *Y* on User *X*'s computer, there is some "leakage" of User *Y*'s data going back 100 seconds. In a fielded system, 40 seconds worth of the data would actually be from User *X* and 60 seconds from User *Y*. However, our experimental setup does not currently support such "mixing" of user behavior. Should a fielded system wish to use W=60 sec, a simple solution would be to average over the last 60 seconds, rather than the last 100 seconds as done in our experiments. We do not expect the impact of such a change to be significant. The data point for W = 10 sec in Figure 1 only uses features that involve no more than the last 10 seconds of measurements, as a reference point – the issue of using less or more than the last 100 seconds of measurements is visited in more depth in the next section.)

To produce Figure 1's results, Table 2's tuning step considered 11 possible settings for $\text{threshold}_{\text{mini}}$ (0.8, 0.85, 0.90, 0.95, 0.97, 1.0, 1.03, 1.05, 1.1, 1.15, and 1.2) and 26 for $\text{threshold}_{\text{full}}$ (0.01, 0.25, 0.5, 0.75, 0.1, 0.125, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.975, 0.99, and 1.0), that is 11x26=286 different combinations of these two parameters. We did not experiment with different choices for the particular values and number of the candidate parameter settings, except as explained in some configurations where we found it necessary to restrict $\text{threshold}_{\text{mini}} = 1.0$, which is the case in Figure 1 for $W = 10$ sec and $W = 60$ sec.

Table 3 shows the highest-weighted features at the end of one experiment, where the weights are averaged over all ten of our experimental subjects and over those values for $W > 10$ used to create Figure 1; for each experimental subject and setting for W, we normalize the weights so that they sum to 1, thus insuring that each configuration contributes equally. Remember that the weights are always changing, so this table should be viewed as a representation "snapshot." (Appendix A contains additional explanations of several of these features). Observe that a wide range of features appear in Table 3: some relate to network traffic, some measure file accesses, others refer to which programs are being used, while others relate to the overall load on the computer. It is also interesting to notice that for some features their average values over 100 seconds are important, whereas for others their instantaneous values matter, and for still others what is important is the *change* in the feature's value. Appendix B displays another list of the highest-weighted features, this time for one specific user.

A weakness of Table 3 is that a measured Windows 2000 property that is important for only one or two subjects might not have a very high average weight. Table 4 provides a different way to see which features play important roles. To produce this table we count how often each measured property appears in the Top 10 weights (including ties, which are common, as can be seen in Appendix B) following training. Surprisingly, over half of the Windows 2000 properties we measure appear at least once in some Top 10 list! This supports our thesis that one should monitor a large number of system properties in order to best create a behavioral model that is well tailored to each individual computer user.

Most of the "derived" calculations (see Section 2.1) are used regularly in the highly weighted features, with the exception of "Difference from Previous Value," which appears in the Top 50 weighted features only about 1/20th as often as the others. "Difference between Current and Average of Last 10" is the most used, but the difference between the most used and the 6th-most used is only a factor of two.

Table 3. Features with the 35 Highest Weights Averaged Across the Experiments that Produced Figure 1

Print Jobs, Average of Previous 100 Values *(ranked #1)*
Print Jobs, Average of Previous 10 Values
System Driver Total Bytes, Actual Value Measured
Logon Total, Actual Value Measured
Print Jobs, Actual Value Measured
LSASS: Working Set, Average of Previous 100 Values
Number of Semaphores, Average of Previous 100 Values
Calc: Elapsed Time,
  Difference between Averages of Prev 10 and Prev 100
Number of Semaphores, Actual Value Measured
LSASS: Working Set, Average of Previous 10 Values
CMD: Handle Count,
  Difference between Current and Average of Last 10
CMD: Handle Count, Average of Previous 10 Values
Write Bytes Cache/sec,
  Difference between Current and Average of Last 10
Excel: Working Set,
  Difference between Current and Average of Last 10
Number of Semaphores, Average of Previous 10 Values
CMD: % Processor Time,
  Difference between Averages of Prev 10 and Prev 100
LSASS: Working Set, Actual Value Measured
System Driver Total Bytes, Average of Previous 100 Values
CMD: % Processor Time,
  Difference between Current and Average of Last 100
CMD: % Processor Time,
  Difference between Current and Average of Last 10
System Driver Resident Bytes, Actual Value Measured
Excel: Handle Count, Average of Previous 10 Values
Errors Access Permissions,
  Difference between Current and Average of Last 10
File Write Operations/sec, Average of Previous 100 Values
System Driver Resident Bytes, Average of Previous 10 Values
System Driver Total Bytes, Average of Previous 10 Values
System Driver Resident Bytes,
  Difference between Current and Average of Last 10
TCP Connections Active, Average of Previous 100 Values
CMD: Working Set,
  Difference between Averages of Prev 10 and Prev 100
CMD: Handle Count,
  Difference between Current and Average of Last 100
Number of Mutexes,
  Difference between Current and Average of Last 10
System Driver Resident Bytes, Average of Previous 100 Values
SYSTEM: Working Set,
  Difference between Current and Average of Last 10
LSASS: % Processor Time,
  Difference between Current and Average of Last 100
Outlook: Handle Count, Average of Previous 100 Values

Table 4. Measurements with the Highest Number of Occurrences in the Top 10 Weights, Including Ties, in the Experiments that Produced Figure 1
(the numbers in parentheses are the percentages of Top 10 appearances)

Number of Semaphores (43%)
Logon Total (43%)
Print Jobs (41%)
System Driver Total Bytes (39%)
CMD: Handle Count (35%)
System Driver Resident Bytes (34%)
Excel: Handle Count (26%)
Number of Mutexes (25%)
Errors Access Permissions (24%)
Files Opened Total (23%)
TCP Connections Passive (23%)
LSASS: Working Set (22%)
LSASS: % Processor Time (22%)
SYSTEM: Working Set (22%)
Notepad: % Processor Time (21%)
CMD: Working Set (21%)
Packets/sec (21%)
Datagrams Received Address Errors (21%)
Excel: Working Set (21%)
MSdev: Working Set (21%)
Server Reconnects (19%)
MSdev: Handle Count (19%)
Write Bytes Cache/sec (18%)
TCP Connections Active (18%)
Write Packets/sec (17%)
UDP Datagrams no port/sec (17%)
WinWord: Working Set (17%)
File Write Operations/sec (16%)
Bytes Received/sec (16%)
Bytes Transmitted/sec (16%)
Read Bytes Paging/sec (16%)
Write Bytes Paging/sec (16%)
Notepad: Elapsed Time (16%)
Powerpnt: Working Set (16%)
File Read Bytes/sec (15%)
File Control Operations/sec (15%)
Packets Received/sec (15%)
TCP Connections Established (15%)
MSaccess: Working Set (15%)
Notepad: Handle Count (15%)
Calc: Elapsed Time (15%)
Bytes Printed/sec (15%)
CMD: % Processor Time (15%)
SYSTEM: Handle Count (15%)
% Total Interrupt Time (14%)
WinWord: Handle Count (13%)
AcroRd32: Elapsed Time (13%)
Outlook: Handle Count (13%)
MSdev: % Processor Time (13%)
TASKMGR: Elapsed Time (13%)
WinZip32: Elapsed Time (13%)
Outlook: Handle Count (13%)
MSdev: % Processor Time (13%)
TASKMGR: Elapsed Time (13%)
WinZip32: Elapsed Time (13%)
Number of Threads (12%)
Number of Sections (12%)
% Physical Memory In Use (12%)
Server Disconnects (12%)
Read Bytes Network/sec (11%)
Number of Open Windows (11%)
File Write Bytes/sec (11%)
% Virtual Memory In Use (11%)
ICMP Messages Received/sec (11%)
ICMP Messages Sent/sec (11%)
Outlook: Working Set (11%)
Explorer: Handle Count (11%)
Excel: % Processor Time (10%)
Files Open (10%)
ICMP Messages/sec (10%)
MSaccess: Handle Count (10%)
Realplay: Elapsed Time (10%)
Errors Logon (8%)
Number of Processes (7%)
Disk Read Bytes/sec (7%)
Outlook: Elapsed Time (7%)
Read Bytes Cache/sec (7%)
UDP Datagrams Sent/sec (7%)
Powerpnt: % Processor Time (7%)
FTP: Handle Count (7%)
UDP Datagrams Received/sec (6%)
WinWord: % Processor Time (6%)
msimn: Handle Count (6%)
msimn: Working Set (6%)
Open Top-Level Windows (5%)
Number of Events (5%)
Notepad: Working Set (5%)
plus another 27 measurements that appeared at least once in the list of top 10 highest weights

3.3 Additional Results and Discussion

In this section we report and discuss experiments involving several variations of our basic approach.

Impact of Using *Relative* Probabilities in the Winnow Algorithm

Recall that in Table 2's Winnow algorithm features "vote" whether or not to sound a (mini)alarm based on the probability of their current value. If this probability is less than some constant *p* (we found *p* = 0.15 works well), then the feature votes to sound an alarm. We have also explored using a variant of this idea. Specifically, we look at the ratio:

prob(***feature=value*** for this user) / prob(***feature=value*** for the general population) [Eq. 2]

An alarm is sounded if this ratio is less than some constant, *r* (we found *r* = 0.33 works well, though just like for *p*, performance appears to be robust to the exact setting of this parameter – values for *r* between 0.25 and 0.75 that we tried worked about the same).

The idea behind using the above ratio is that it focuses on feature values that are rare for this user relative to their probability of occurrence in the general population. For example, feature values that are rare for User *X* but also occur rarely across the general population may not produce low ratios, while feature values that are rare for User *X* but are not rare in general will. That is, this ratio distinguishes between "rare for User *X* and for other computer users as well" and "rare for User *X* but not rare in general."

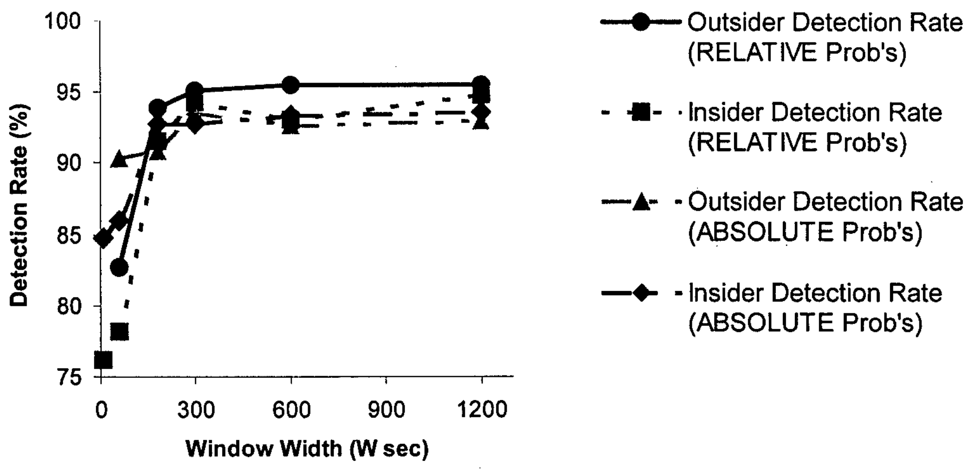
We estimate probability(***feature=value*** for the general population) by simply pooling all the data from the ten "insider" experimental subjects, and then creating a discrete probability distribution using ten bins, using the technique explained earlier. Doing this in a fielded system would be reasonable, since in our IDS design one already needs a pool of users for the training and tuning phases.

Figure 2 reports the test-set results of using the two different ways of deciding when to raise an alarm. When using individual probabilities and the threshold *p* we say we are using "absolute" probabilities and when we are using the ratio of probabilities (Eq. 2 above) and the threshold *r* we say we are using "relative" probabilities. Notice that using the relative probabilities produces slightly better detection rates on the testing data.

Because our experimental setup only involves measurements from normal computer users, the use of our ratio of probabilities makes sense in our experiments, since it defines "rare for User *X*" relative to the baseline of other computer users operating normally. However, it is likely that the behavior of intruders, even insiders working at the same facility, may be quite different from normal computer usage (unfortunately we do not yet have such data to analyze). For example, an intruder might do something that is rare in general, and hence Equation 1 above might not produce a value less than the setting for the threshold *r*.

This argument suggests that *both* relative and absolute probabilities should be used in the Winnow algorithm (Table 2). We have done a preliminary experiment where a mini-alarm (Step 2b of Table 1) is sounded if *either* the absolute or relative versions of the Winnow algorithm sound an alarm. In this experiment (using W=1200 sec), the test-set detection rate was 97.3% for this combined approach versus 94.7% for "relative-only" approach and 93.6% for the "absolute-only" version; this detection rate is achieved with less than one false alarm per day per user (one false alarm every *three* days actually, which is the same as in the "absolute-only" version, while the "relative-only" version has one false alarm about every four days).

Figure 2. Using Absolute vs. Relative Probabilities to Trigger Mini-Alarms

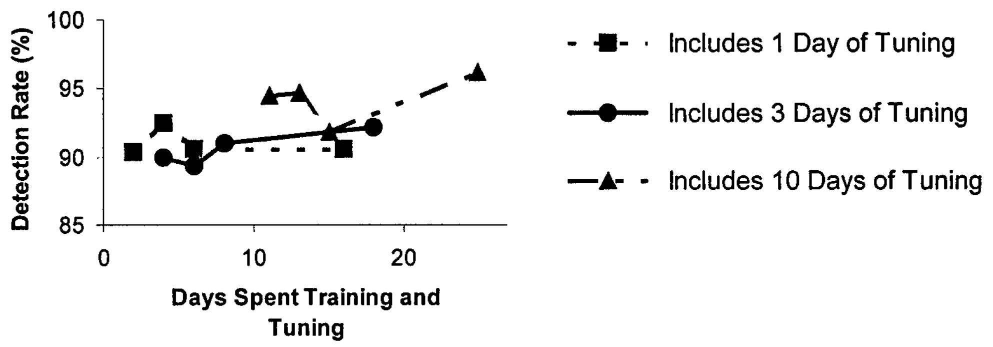


Impact of the Amount of Training and Tuning Data Used

In the experiments reported so far, we have been using three weeks of training data, two weeks of tuning, and two weeks of testing data per user. In this section we evaluate the impact of spending less time training and tuning; the length of the testing period is less relevant since its role is to represent future behavior after the IDS's have been trained. Figure 3 reports the detection rate on the test data, averaged over the ten experimental subjects, for various configurations of training and tuning durations (as in all our curves, in all cases our specified false-alarm rate on the testing data is met). "Days" in this figure refers to "days of data" and not "days of CPU time." Each data point is the average of the results obtained when using "relative" and "absolute" probabilities (in two separate runs) to trigger mini-alarms. To create this figure we use only three days of testing data per user. In Figure 3 the right-most "diamond" marks the results that correspond to those in Figure 2, except here the test set contains 1/3rd as much data (and this smaller test set seems slightly easier than the two-week-long version, possibly because users' behavior varies less over a shorter period of time).

There is a slightly increasing trend in detection rate on the testing data as more time is spent training and tuning, but even with only one day of training and one of tuning, performance is good. This suggests that a system can be fielded after only a very short training period; in addition, performance of our algorithm is likely to increase as additional measurements are gathered (since we continually retrain).

Figure 3. Detection Rates as Function of Training Period (W = 300 sec)



We have found that unless more than a week of data is used for tuning, Table 1's $\text{threshold}_{\text{mini}}$ should be constrained to be equal to 1 and Table 1's algorithm should only tune the value of $\text{threshold}_{\text{full}}$. Otherwise there are too many degrees of freedom and not enough tuning data, which leads to over fitting the tuning data and poor performance on the testing data. More specifically, when less than a week of tuning data is used, the resulting false-alarm rate on the testing data occasionally exceeds our goal of one per day per user, even though in its tuning phase Table 1's algorithm can find setting for the two thresholds that produce *no* false alarms on the tuning dataset.

When there is sufficient tuning data (e.g., two weeks worth), tuning both parameters occasionally leads to some improvement in detection rates on the testing data. However, it is not clear that tuning $\text{threshold}_{\text{mini}}$ is beneficial, due to the increased risk of over fitting, and for a fielded system it would be reasonable to solely tune $\text{threshold}_{\text{full}}$.

Impact of the "Memory Length" of the Derived Features

Table 3 shows that the features that use the last *N* measurements of a Windows 2000 property play an important role. Figure 4 illustrates the performance of Table 1's algorithm when we use features that use at most the last 1, 10, 100, or 1000 measurements, respectively, of the Window 2000 properties (Appendix A) we monitor. The Y-axis is the test-set detection rate and in all cases the false-alarm rate meets our goal of no more than one per user per workday. Figure 4's data is from the case where W = 300 seconds; 15 days of training data, 3 of tuning, and 3 of testing are used for each experimental subject.

Figure 4. Detection Rate as Function of Number of Previous Values Used (W = 300 sec)

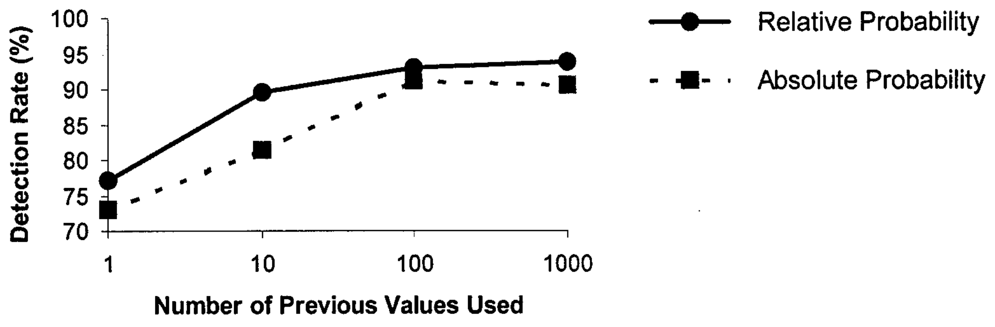


Figure 4 shows that there is an advantage in considering features that have longer "histories." However, the cost of a longer history is that more data needs to be collected to define a feature value. That is, if histories can go back as far as 1000 seconds (a little over 15 minutes), then it will take 1000 seconds after an intrusion until all of the feature values are due solely to the intruder's behavior. It appears that limiting features to at most the last 100 seconds of measurements is a good choice (a minor future research task would be to more finely sample the X-axis of Figure 2, rather than only using history lengths that are powers of ten).

Impact of the Number of Users in the Training Pool

The experiments reported so far always involved ten subjects in the pool of "insiders" used during training. The question naturally arises as to the impact of having smaller (or larger) local workgroups of "insiders." Figure 5 presents test-set results that partially address this question. Three times we trained and tuned using random subsets of five and of seven of our pool of ten insiders; in this experiment we only tuned with only one day's worth of data and also only tested with a (different) day's worth of data. We also evaluated our learned models in these six experiments using our standard set of "outsiders" data. As is standard in our experiments, for all of the results in Table 5, the false-alarm rate is less than one per day per user.

Figure 5. Detection Rate as Function of Number of Members of the Local Work Group (W = 300 sec)

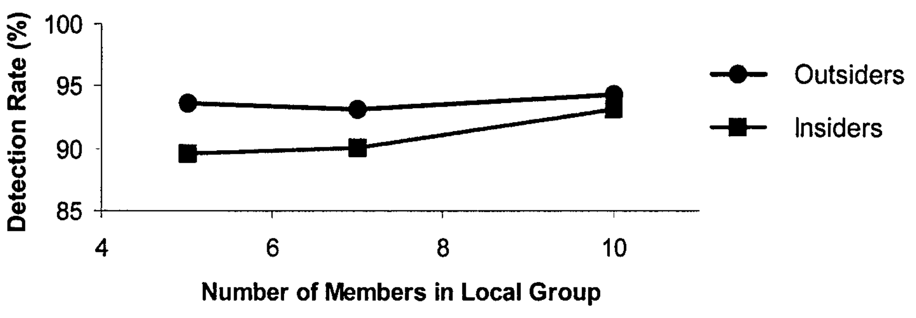


Figure 5 indicates that there is a slight tendency to do better as the number of users in the local work group increases, while the detection rate on the outsiders is approximately constant. Our initial hypothesis was that it would be easier to learn to detect insider intrusions with smaller numbers of insiders, but that detecting outsider intrusions would be harder, since the learning algorithm had to deal with fewer types of behavior during the training process. However, the experimental results do not match this hypothesis. It is also unclear why there is such a large gap between the detection rates for insiders and outsiders on the left half of Figure 5. One way to resolve these mismatches is to perform a larger experiment, say with ten times as many subjects, and it certainly is the case that an "insiders" pool containing only ten users is not very large for "scaling up" experiments.

Tables 5 and 6 that are described in the next section show that the individual variance in intrusion-detection rates is high; hence, experiments with small numbers of subjects can be greatly impacted by the particular make-up of their subject pools. We include the results of Figure 5 in this final report not because they provide any deep insights into the strengths and weaknesses of Table 1's algorithm, but rather as an illustration of what kinds of experiments should be done with this algorithm if data can be obtained from a larger number of people.

Individual Differences for Table 1's Algorithm

So far we have reported results average over our pool of 10 insiders and 6 outsiders. It is interesting to look at results from individual experimental subjects. Table 4 reports how often User *Y* was *not* detected when "intruding" on User *X*'s computer. For example, the cell <*row=User6, column=U5*> says that the probability of detection is 0.35 when User 5 operates on User 6's computer for 1200 seconds. (The rightmost column is the detection rate when outsiders operate on each insider's computer.)

Table 5. Fraction of Times that User Y Successfully Intruded on User X's Machine (using *absolute* probabilities and W = 1200 sec)

	Y = User 0	U1	U2	U3	U4	U5	U6	U7	U8	U9	RowAve	Outs
X=User0		**34%**	**40%**	0%	**53%**	0%	0%	0%	**37%**	0%	**19%**	**11%**
User1	1%		2%	1%	0%	0%	0%	0%	0%	3%	1%	0%
User2	7%	**78%**		0%	1%	1%	0%	0%	1%	**24%**	**12%**	0%
User3	0%	0%	0%		0%	0%	0%	0%	0%	1%	0%	5%
User4	**100%**	0%	8%	0%		0%	0%	0%	0%	0%	**16%**	7%
User5	0%	0%	0%	0%	0%		0%	0%	4%	0%	0%	2%
User6	0%	0%	0%	0%	0%	**35%**		0%	8%	6%	6%	**12%**
User7	0%	0%	0%	0%	0%	0%	0%		2%	0%	0%	0%
User8	0%	0%	0%	0%	0%	0%	0%	0%		0%	0%	0%
User9	0%	4%	2%	0%	0%	**12%**	0%	0%	0%		2%	4%
ColAve =	**12%**	**13%**	6%	0%	6%	5%	0%	0%	6%	4%	5%	4%

Table 6. Fraction of Times that User Y Successfully Intruded on User X's Machine (using *relative* probabilities and W = 1200 sec)

	Y = User 0	U1	U2	U3	U4	U5	U6	U7	U8	U9	RowAve	Outs
X=User0		0%	7%	0%	**23%**	0%	0%	0%	**41%**	0%	7%	**15%**
User1	5%		1%	0%	**35%**	0%	0%	0%	0%	2%	6%	0%
User2	0%	0%		0%	**26%**	0%	0%	0%	0%	5%	4%	0%
User3	**14%**	0%	0%		0%	0%	0%	0%	0%	0%	2%	0%
User4	2%	0%	4%	0%		0%	0%	0%	0%	4%	4%	0%
User5	0%	0%	0%	0%	0%		0%	0%	1%	0%	0%	**20%**
User6	0%	0%	0%	0%	0%	0%		0%	**45%**	**75%**	**14%**	2%
User7	0%	0%	0%	0%	0%	0%	0%		0%	0%	0%	4%
User8	0%	0%	0%	0%	0%	0%	0%	3%		0%	0%	2%
User9	9%	**60%**	**16%**	1%	0%	**43%**	6%	0%	**25%**		**17%**	3%
ColAve =	3%	7%	5%	0%	9%	5%	1%	0%	**12%**	**10%**	5%	5%

Given that the overall detection rate is about 95% (i.e., only 5% of 1200-sec intrusions do not sound alarms), one might expect that most of the individual penetration rates would range from, say, 2% to 10%. However, the results are much more skewed. In most cases, *all* the attempted intrusions are detected – the majority of cells in Table 5 contains 0's (in fact we report "penetration" rates rather than detection rates in this table because otherwise all of the 100%'s would be visually overwhelming). But in several cases (highlighted with bold font) a user is frequently not detected when operating on another user's computer.

Table 6 repeats Table 5, with the only difference being that the results were produced using *relative* probabilities in the Winnow algorithm. Interestingly, the easily confused user-pairs are generally very different in the two cases (cells that have high values in both tables are underlined in Table 6); recall that earlier in this section we present results of a successful combined approach that uses both the absolute and relative probabilities.

One implication of the results in Tables 5 and 6 is that one could run experiments like these on some group of employees, and then identify for which ones their computer behavior is sufficiently distinctive that Table 1's algorithm provides them effective protection. For example, Users 1, 3, 7, and 8 in Tables 5 and 6 detect almost all of the intrusion attempts on their computers.

Comparison to the Naïve Bayes Algorithm

A successful algorithm on many tasks is the Naïve Bayes algorithm [MITCHELL97]. We applied this algorithm in the same experimental setup as used to evaluate Table 1's algorithm. However, the best results we have been able to obtain (for W = 1200 seconds) are a 59.2% detection rate with an average of 2.0 false alarms per day per user, which compares poorly to Table 1's algorithm's results, in the identical scenario, of a 93.6% detection with an average of 0.3 false alarms per day per user. Nevertheless, there are some aspects of the Naïve Bayes algorithm for which it makes sense to consider integrating into Table 1's algorithm. This topic is further discussed in Section 5, Future Work.

4. Experiments with Keystroke Monitoring

Section 3's experiments did not involve any direct measurements of user's typing behavior. In Section 4 we investigate how well individual differences in typing behavior can play an identifying role. Monrose and Rubin [MONROSE97] performed similar studies that investigated using keystroke dynamics as an authentication mechanism, but here we use keystroke behavior as an aid for intrusion detection.

We implemented an experimental approach that works as follows:

1. Use 50,000 keystrokes for each user to *train* a separate statistical model for each user. (On average, users type 5000 keystrokes per working day, so this is about two weeks of activity.)

2. Use another 50,000 keystrokes to *tune* some parameters (explained further below), separately for each user. The tuning data is collected from non-overlapping days than those used to create the training data. (Similarly, the testing data described below also comes from days separate from those where the training and tuning data were collected.)

3. Use a third set of 30,000 keystrokes as a *test* set. As emphasized in Section 3, it is extremely important for proper experimental methodology that one uses *separate* data to tune parameters and to estimate future accuracy. "Tuning on the test set" will usually lead to overestimates of future accuracy.

The basic algorithm we developed works as follows:

1. Compute the probability of the last three keystrokes (including the time taken between keystrokes and the time each key was held down before being released), given the model learned for the given machine's normal user. The specific probability that we measure is explained below.

2. If this probability is lower than some threshold, T, then "mark" this keystroke.

3. If there are more than N marks in the last W keystrokes, raise an alarm.

Our algorithm automatically chooses the T and N settings for each person and for each W, based on optimizing accuracy on the *tuning* data set that was mentioned above, while holding the rate of false alarms to less than one per work day per user (actually, less than one per 5000 keystrokes, since that is the average number of keystrokes our users typed per day).

The probability that we measure is based on the three previous keystrokes and timings related to them:

Prob(	current keystroke	= *Key3*	and
	previous keystroke	= *Key2*	and
	two-ago keystroke	= *Key1*	and
	time between *K2* and *K3*	= *Interval23*	and
	time between *K1* and *K2*	= *Interval12*	and
	time *K3* was down	= *Downtime3*)	

K3 is the most recent keystroke, K2 the middle keystroke, and K1 the earliest.
If there has not been a new keystroke in the last two seconds an "idle" keystroke is inserted, and we ignore in our experiments all sequences of three successive "idles."

When an experimental subject has not typed anything for 30 minutes, the buffer containing the last W keystrokes is cleared and no diagnosis will be made until a fresh run of W keystrokes occurs.

We compressed the possible keystrokes into 13 groups (lower-case alphabetic; upper-case alphabetic; numeric; F-key; white space; left-hand shift, control, or alt; right-hand shift, control, or alt; backspace and delete; numeric keypad; punctuation; math symbol; "other;" and idle) for keystroke 3, eight groups for keystroke 2, and three for keystroke 1 (idle, alphabetic, and other). We reduce the number of groups for the earlier keystrokes in order to reduce the size of the joint probability distribution that we need to estimate.

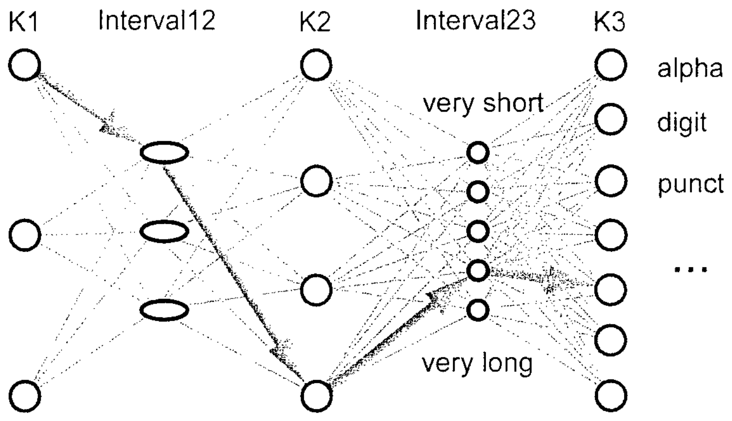
The above probability formula requires inter-key times and key-down times to be discretized. We discretized the time between keystroke 2 and keystroke 3 into these five bins: ≤ 20 msec, 20-80 msec, 80-150 msec, 150-750 msec, and > 750 msec. We discretized the time between keystroke 1 and keystroke 2 into three bins: ≤ 80 msec, 80-750 msec, and > 750 msec. We discretized key-down times into four bins: ≤ 35 msec, between 35-75 msec, 75-175 msec, and > 175 msec. We choose these thresholds by creating histograms using some of the training data and then visually inspecting these histograms.

Hence each user's probability table involves $13 \times 8 \times 3 \times 5 \times 3 \times 4 = 18{,}720$ cells. We initialize all of these cells to 1 (to insure that we never estimate probabilities of zero) and then simply fill each user's probability table by processing that user's training data and counting how often each possible combination occurs.

Figure 6 illustrates the probability we compute (though, to reduce clutter, the down time for keystroke 3 is not shown). Each node represents a possible value for one of the random variables (K1, Interval12, etc) appearing in the probability that we estimate. A path through this graph corresponds to an entry in our large joint probability distribution; the path with thick lines and arrow heads is one such path, where, say, the user first pressed the letter 'a', quickly followed by pressing the left control key, and then after 500 msec pressed the F5 key. Our probability table simply contains the estimated probability of following each possible path, going left-to-right, through this graph.

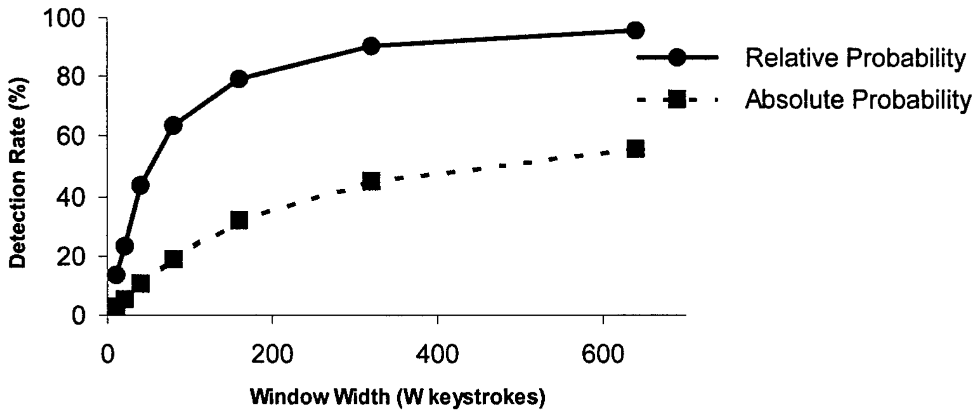
Our current accuracy results on the *test* data set, as a function of W, appear in Figure 7. When measuring accuracies on the testing data, we use non-overlapping windows (of W keystrokes) in order to reduce the correlation between successive samples.

Figure 6. Visualizing the Three-Keystroke Probability



Analogous to our use of both absolute and relative probabilities in Section 3.3, here we consider both using directly the probability illustrated by Figure 5 and also using the ratio of User *X*'s probability of producing the last three keystrokes (including timing) divided by the general population's probability of doing so. For keystroke analysis, looking at relative probabilities is clearly greatly advantageous. This makes sense when one considers that some keystroke combinations – for example, lower-case letter, followed by backspace, and then followed by a capital letter – are rare for most users and it seems reasonable to normalize with respect to "background" rates across the population.

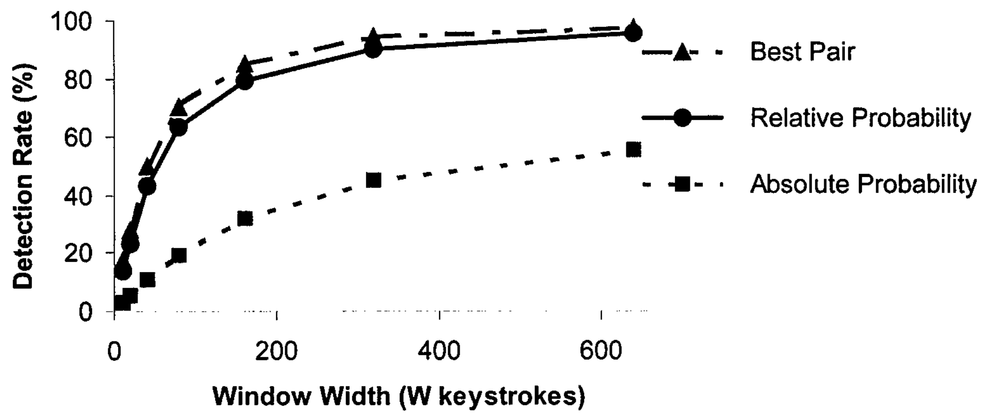
Figure 7. Detection Rates of Keystroke Analysis



We are encouraged that we can recognize a sizable fraction of intrusions (defined as User *X* typing on user *Y*'s computer) which such a low false-alarm rate, especially since we are currently only using in these experiments one type (i.e., keystrokes) of the many types of data we have collected. Of course an intruder can do a lot of damage in, say, 160 keystrokes, but we believe that detecting with large windows can still be useful; it certainly is better than looking at yesterday's log files.

Recall that our algorithm chooses the best parameter settings (T and N). We have expended this to find the optimal *pair*, T_1 & N_1 and T_2 & N_2, during the tuning process. If *either* settings recommend raising an alarm, then an alarm is raised. Figure 8's top-most line shows the performance of this version (the other two lines are repeated from Figure 7). This variant produces an improvement of 2-7 percentage points in the intrusion-detection rate on the test set.

Figure 8. Detection Rate Using the Best Pair of Detectors



A solution to the above-mentioned weakness of having to wait W keystrokes before an alarm can be sounded, is to have a *collection* of IDS's for each user, where each individual IDS uses a different value for W – say 10, 20, 40, 80, 160, and 320 seconds. One would need to carefully choose which and how many individual IDS's to combine in order to insure that the full ensemble did not produce too many false alarms, but the advantage is that intrusions would be detected as early as possible. We have performed some very preliminary experiments, whose results however are not described in this report, that indicate that such an ensemble-of-IDS's approach can work well.

We have not yet combined any keystroke-based "intrusion detector" with the algorithm of Sections 2 and 3; doing so is a topic for future research. One way to accomplish this would be to use the probability described in this section as one more feature in the Winnow algorithm.

5. Future Work

Before concluding the final report on out ATIAS project, we discuss a few possible extensions to the work reported above that have not yet been fully discussed. An obvious extension is to obtain and analyze data from a larger number of users, as well as data from a collection of server machines. And of course it would be greatly beneficial to have data gathered during actual intrusions, rather than simulating them by replaying one user's measurements on another user's computer. Among other advantages, having data from a larger pool of experimental subjects would allow "scaling up" issues to be addressed, statistically justified confidence intervals on results to be produced, and parameters to be better tuned (including many for which we have "hard-wired in" values in our current experiments).

When we apply the Winnow algorithm during the training phase (Step 1 in Table 1), we get remarkable accuracies. For example, out of 3,000,000 seconds of examples (half that should be called an intrusion and half that should not), we consistently obtain numbers on the order of only 150 missed intrusions and 25 false alarms, and that is from starting with all features weighted equally. Clearly the Winnow algorithm can quickly pick out what is characteristic about each user and can quickly adjust to changes in the user's behavior. In fact, this rapid adaptation is also somewhat of a curse (as previously discussed in Section 2), since an intruder who is not immediately detected may soon be seen as the normal user of a given computer. This is why we look for N mini-alarms in the last W seconds before either sounding an alarm or calling the recent measurements normal and then applying Winnow to these measurements; our assumption is that when the normal user changes behavior, only a few mini-alarms will occur, whereas for intruders the number of mini-alarms produced will exceed N. Nevertheless, we still feel that we are not close to fully exploiting the power of the Winnow algorithm on the intrusion-detection task. With more tinkering and algorithmic variations, it seems possible to get closer to 99% detection rates with very few false alarms.

In Section 2's Winnow-based algorithm we estimate the probability of the current value for a feature and then make a simple "yes-no" call (see Eq. 1), regardless of how close the estimated probability is to the threshold. However, it seems that an extremely low probability should have more impact than a value just below the threshold. In the often-successful Naïve Bayes algorithm, for example, actual probabilities appear in the calculations, and it seems worthwhile to consider ways of somehow combining the weights of Winnow and the actual (rather than thresholded) probabilities.

In our main algorithm (Table 1) we did not "condition" the probabilities of any of the features we measured. Doing so might lead to more informative probabilities and, hence, better performance. For example, instead of simply considering Prob(File Write Operations/sec), it might be more valuable to use Prob(File Write Operations/sec | MS Word is using most of the recent cycles), where '|' is read "given." We have done some preliminary work on choosing the best two conditions for each of our features, but deferred that work when we discovered how well the Winnow algorithm (which was not part of our project proposal) worked.

In none of the experiments of this report did we *mix* the behavior of the normal user of a computer and an intruder, though that is likely to be the case in practice. It is not trivial to combine two sets of Windows 2000 measurements in a semantically meaningful way (e. g., one cannot simply add the two values for each feature or, for example, CPU utilizations of 150% might result). However, with some thought it seems possible to devise a plausible way to mix normal and intruder behavior. An alternate approach would be to run our data-gathering software while someone is trying to intrude on a computer that is simultaneously being used by another person.

In the results reported in Section 3, we tune parameters to get zero false alarms on the tuning data, and we found that on the testing data we were able to meet our goal of less than one false alarm per user per day (often we obtained test-set results more like one per week). If one wanted to obtain even fewer false alarms, then some new techniques would be needed, since our approach already is getting no false alarms on the tuning set. One solution we have explored is to tune the parameters to zero false alarms, and then to increase the stringency of our parameters - e. g., require 120% of the number of mini-alarms as needed to get zero tuning-set false alarms. More evaluation of this and similar approaches is needed.

We have also collected Windows 2000 event-log data from our set of 16 Shavlik Technologies employees. However we decided not to use that data in our experiments since it seems one would need to be using data from people actually trying to intrude on someone else's computer for interesting event-log data to be generated. Our approach for simulating "intruders" does not result in then generation of meaningful event-log entries like failed logins. We also collected mouse-movement data from users, but due to some initial technical difficulties we did not obtain that until late in the project and did not have time to use it to augment the keystroke-analysis experiments of Section 4.

Another type of measurement that seems promising to monitor are the *specific* IP addresses involved in traffic to and from a given computer. Possibly interesting variables to compute include the number of different IP addresses visited in the last N seconds, the number of "first time visited" IP addresses in the last N seconds, and differences between incoming and outgoing IP addresses.

A final possible future research topic is to extend the approaches in this report to local *networks* of computers, where the statistics of behavior across the set of computers is monitored. Some intrusion attempts that might not seem anomalous on any one computer, may appear highly anomalous when looking at the behavior of a set of machines.

6. Conclusion

The goal of this project is to continually gather and analyze hundreds of fine-grained measurements about Windows 2000 system performance, such as network traffic, identity of the current programs executing, and the user's typing behavior (a full list appears in Appendix A). Our scientific hypothesis is that a properly chosen set of measurements can provide a "fingerprint" that is unique to each user, serving to accurately distinguish appropriate use of a given computer from misuse.
Section 3's and 4's empirical evaluation of the algorithms that we developed indicate it is possible to accurately distinguish between the normal use by the owner of a given computer and use by someone else. We also provide some insights into which system measurements play the most valuable roles in creating statistical profiles of users (Tables 3 and 4, plus Appendix B). We have been able to get high intrusion-detection rates (95%) and low false-alarm rates (less than one per day per computer) without "stealing" too many CPU cycles (less than 1%). We believe it is of particular importance to have very low false-alarm rates; otherwise the warnings from IDS will soon be disregarded. Our project demonstrates that computer security can be enhanced by monitoring each user's (or server's or intelligent software agent's) behavior, learning statistical models based on these measurements, and then using these statistical models to accurately detect anomalous behavior, which might be indicative of insider misuse.

Specific key lessons learned in this project are that it is valuable to:

- consider a large number of different properties to measure, since many different features play an important role in capturing the idiosyncratic behavior of at least some user (see Table 4)
- continually reweight the importance of each feature measured (since users' behavior changes)
- look at features that involve more than just the instantaneous measurements (e. g., averages over the last 100 seconds, differences between the current measurement and the average over the last 10 seconds – see Figure 4)
- tune parameters on a per-user basis (e. g., the number of "mini alarms" in the last *N* seconds that are needed to trigger an actual alarm)
- tune parameters on "tuning" datasets and then estimate "future" performance by measuring detection and false-alarm rates on a separate "testing" set (if one only looks at performance on the data used to train and tune the learner, one will get unrealistically high estimates of future performance; for example, we are always able to tune to *zero* false alarms)
- look at the variance in the detection rates across users; for some, there are no or very few missed intrusions, while for others a sizable number of intrusions are missed (see Tables 5 and 6) – this suggests that for at least some users (or servers) our approach can be particularly highly effective

An anomaly-based IDS should not be expected to play the sole intrusion-detection role, but such systems nicely complement IDS that look for known patterns of abuse. New misuse strategies will always be arising, and anomaly-based approaches provide an opportunity to detect them even before the internal details of the latest intrusion strategy is fully understood.

As final comment, we wish to note that the algorithms presented in this report do not apply solely to Windows 2000 measurements gathered on personal workstations. Other than the keystroke-analysis algorithm of Section 4, our approach directly applies to Windows-based servers. And to apply to other operating systems or even to specific applications (e.g., a Java-based logistics planner, intelligent software agents, or some important database program) one need only adapt the code that measures specific system (or application) properties periodically; the data-analysis and model-building algorithms can then be applied directly to such data.

Acknowledgements

We wish to thank the employees of Shavlik Technologies who volunteered to have data gathered on their personal computers. (Although the title of this project mentions Windows NT, by the time the project was funded, all of these employees have upgraded to Windows 2000. However, the changes in the properties monitored are minor between NT and 2000, and the results in this report should accurately reflect performance on Windows NT, as well as Windows XP, computers.)

Appendix A – List of Windows 2000 Properties Measured

This appendix reports the 207 Windows 2000 properties that we measure. Additional documentation about many of them is available by running Performance Monitor (perfmon) in Windows 2000.

```
Number of Open Windows            // Number of windows open.
Open Top-Level Windows            // Number of windows open, not counting children.

File Read Operations/sec
File Read Bytes/sec
File Write Operations/sec
File Write Bytes/sec
File Control Operations/sec

Context Switches/sec
System Calls/sec

% Total Processor Time
% Total User Time                 // % of non-idle processor time spent in user mode
% Total Interrupt Time
% Total Privileged Time

Total Interrupts/sec
Data Maps/sec

Number of Processes
Number of Threads
Number of Events
Number of Semaphores
Number of Mutexes
Number of Sections

Cache Faults/sec
Pages/sec

System Driver Total Bytes         // Bytes of virtual memory in use by device drivers
System Driver Resident Bytes      // Working set of the above
% Physical Memory In Use
% Virtual Memory In Use

Bytes Received/sec
Bytes Transmitted/sec

Errors Logon
Errors Access Permissions         // Count of illegal file accesses
Errors Granted Access             // Count of denied accessed to files later opened

Files Opened Total                // Files opened by system
Files Open                        // Number of files current open

Server Sessions
File Directory Searches
Pool Paged Failures               // Number of times allocations from paged pool failed
```

```
Logon/sec
Logon Total

Bytes Total/sec
File Data Operations/sec
Packets/sec
Packets Received/sec
Packets Transmitted/sec

Read Bytes Paging/sec
Read Bytes Non-Paging/sec
Read Bytes Cache/sec
Read Bytes Network/sec
Read Packets/sec
Reads Denied/sec

Write Bytes Paging/sec
Write Bytes Non-Paging/sec
Write Bytes Cache/sec
Write Bytes Network/sec
Write Packets/sec
Writes Denied/sec

Disk Reads/sec
Disk Writes/sec
Disk Transfers/sec
Disk Read Bytes/sec
Disk Write Bytes/sec
Disk Bytes/sec

Network Errors/sec
Server Reconnects              // Number of active sessions that are timed out
Connects Windows NT            // Number of connections to Windows NT computers
Server Disconnects             // Number of times Redirector disconnected
Server Sessions Hung

TCP Connections Active
TCP Connections Passive
TCP Segments/sec
TCP Connections Reset
TCP Segments Received/sec
TCP Segments Sent/sec
TCP Connections Established

Datagrams Received Address Errors
Datagrams Forwarded/sec
Datagrams Received Header Errors
Datagrams/sec
Datagrams Sent/sec
Datagrams Received/sec
Datagrams Received Unk Prot

UDP Datagrams/sec
UDP Datagrams Sent/sec
UDP Datagrams Received/sec
UDP Datagrams no port/sec
UDP Datagrams Received errors

ICMP Messages Received/sec
ICMP Messages Sent/sec
ICMP Messages/sec

Current Commands              // Number of commands in Redirector queue
Total RAS Connections         // RAS = Remote Access Server

Current Outgoing Phone Calls
Current Incoming Phone Calls

Bytes Printed/sec
Number of Print Jobs          // Current number of jobs in a print queue

_total: % Processor Time      // Summed over all processes
_total: Handle Count
_total: Working Set

// The remaining measurements relate to commonly run Windows 2000 programs
WinWord: % Processor Time
WinWord: Handle Count
WinWord: Working Set

Excel: % Processor Time
Excel: Handle Count
Excel: Working Set

Powerpnt: % Processor Time
Powerpnt: Handle Count
Powerpnt: Working Set

MSaccess: % Processor Time
MSaccess: Handle Count
MSaccess: Working Set

Outlook: % Processor Time
Outlook: Handle Count
Outlook: Working Set
Outlook: Elapsed Time

msimn: % Processor Time       // Outlook express
msimn: Handle Count
msimn: Working Set

Notepad: % Processor Time
Notepad: Handle Count
Notepad: Working Set
Notepad: Elapsed Time

Wordpad: % Processor Time
Wordpad: Handle Count
Wordpad: Working Set
WordPad: Elapsed Time
```

```
MSdev: % Processor Time
MSdev: Handle Count
MSdev: Working Set

Explorer: % Processor Time
Explorer: Handle Count
Explorer: Working Set

IExplorer: % Processor Time     // Web browers
IExplorer: Handle Count
IExplorer: Working Set
Netscape: % Processor Time
Netscape: Handle Count
Netscape: Working Set

Eudora: % Processor Time        // A popular mail program
Eudora: Handle Count
Eudora: Working Set

vb5: % Processor Time           // Visual Basic
vb5: Handle Count
vb5: Working Set
vb6: % Processor Time
vb6: Handle Count
vb6: Working Set

jview: % Processor Time         // Java executors
jview: Handle Count
jview: Working Set
wjview: % Processor Time
wjview: Handle Count
wjview: Working Set
java: % Processor Time
java: Handle Count
java: Working Set

notes: % Processor Time         // Lotus notes
notes: Handle Count
notes: Working Set

SPOOLSS: % Processor Time       // The print spooler
SPOOLSS: Handle Count
SPOOLSS: Working Set

RPCSS: % Processor Time         // PRC = remote procedure call
RPCSS: Handle Count
RPCSS: Working Set

LSASS: % Processor Time         // LSASS = Local Security Authority SubSystem
LSASS: Handle Count
LSASS: Working Set

TCPSVCS: % Processor Time       // TCP server
TCPSVCS: Handle Count
TCPSVCS: Working Set
```

```
AT: % Processor Time                // Runs commands AT a specified time
AT: Handle Count
AT: Working Set

CMD: % Processor Time               // The Windows command-line interpreter
CMD: Handle Count
CMD: Working Set

COMMAND: % Processor Time           // Command.com
COMMAND: Handle Count
COMMAND: Working Set

FINDSTR: % Processor Time           // Searches for strings in files
FINDSTR: Handle Count
FINDSTR: Working Set

FINDFAST: % Processor Time          // Used to index Microsoft Office documents
FINDFAST: Handle Count
FINDFAST: Working Set

FTP: % Processor Time               // FTP = file transfer protocol
FTP: Handle Count
FTP: Working Set
FTP: Elapsed Time

PRINT: % Processor Time             // Prints a text file
PRINT: Handle Count
PRINT: Working Set
PRINT: Elapsed Time

CONTROL: % Processor Time
CONTROL: Handle Count
CONTROL: Working Set

SYSTEM: % Processor Time            // A kernel process
SYSTEM: Handle Count
SYSTEM: Working Set

// See how much cpu time has been expended by various other programs.
Calc: Elapsed Time                  // Windows' built-in calculator
TASKMGR: Elapsed Time               // Task manager
QuickTimePlayer: Elapsed Time       // Apple's Quicktime media player
Mplayer2: Elapsed Time              // Microsoft's media player
Realplay: Elapsed Time              // Real Audio's media player
AcroRd32: Elapsed Time              // Adobe's viewer of PDF
WinZip32: Elapsed Time              // Compress and uncompress files
```

Appendix B – The Top 50 Measurements for One User

Below are the top fifty (plus a few more due to ties) weighted features for a sample user, who is a secretary and is User #4 in Tables 5 and 6. These results are from the scenario of using *relative* probabilities (see Section 3.3). Also, unlike Table 3, in this appendix a measured Windows 2000 property appears at most *once* in order to increase readability; whichever derived feature has the most weight is reported, with ties being broken in favor of derived features that appear earliest in Section 2's list of derived features.

Best #1

File Control Operations/sec
  Average of Previous 100 Values
Bytes Received/sec
  Average of Previous 10 Values
Number of Print Jobs
  Average of Previous 10 Values
Excel: Working Set
  Difference between Current and Average of Last 10
LSASS: Working Set
  Average of Previous 100 Values
CMD: Handle Count
  Difference between Current and Average of Last 100

Best #7 (weight = highest weight / 2)

Number of Semaphores
  Average of Previous 10 Values
Errors Access Permissions
  Average of Previous 10 Values
TCP Segments Received/sec
  Average of Previous 100 Values
UDP Datagrams no port/sec
  Difference between Current and Average of Last 100
_total: Handle Count
  Average of Previous 100 Values
Excel: Handle Count
  Difference between Current and Average of Last 100
Powerpnt: % Processor Time
  Average of Previous 100 Values
LSASS: Handle Count
  Difference between Current and Average of Last 10
TASKMGR: Elapsed Time
  Average of Previous 10 Values

Best #16 (weight = highest weight / 4)

Number of Open Windows
  Difference between Current and Average of Last 100
% Total Processor Time
  Difference from Previous Value
% Total Interrupt Time
  Average of Previous 100 Values
Number of Threads

```
    Average of Previous 10 Values
  Write Packets/sec
    Average of Previous 100 Values
  Powerpnt: Handle Count
    Average of Previous 10 Values
  Outlook: % Processor Time
    Average of Previous 10 Values
  Outlook: Working Set
    Difference between Current and Average of Last 10
  CMD: Working Set
    Average of Previous 10 Values
  SYSTEM: Working Set
    Actual Value Measured
  AcroRd32: Elapsed Time
    Difference between Current and Average of Last 10
```

Best #27 (weight = highest weight / 8)

```
  % Physical Memory In Use
    Average of Previous 10 Values
  TCP Connections Reset
    Average of Previous 10 Values
  Datagrams/sec
    Average of Previous 10 Values
  Explorer: Handle Count
    Difference between Current and Average of Last 10
  Explorer: Working Set
    Difference between Averages of Prev 10 and Prev 100
```

Best #32 (weight = highest weight / 16)

```
  Open Top-Level Windows
    Difference between Current and Average of Last 10
  Number of Events
    Difference between Averages of Prev 10 and Prev 100
  Number of Mutexes
    Average of Previous 10 Values
  Cache Faults/sec
    Average of Previous 10 Values
  System Driver Resident Bytes
    Average of Previous 10 Values
  % Virtual Memory In Use
    Difference between Current and Average of Last 10
  Files Opened Total
    Average of Previous 100 Values
  Connects Windows NT
    Difference between Current and Average of Last 10
  Datagrams Received Address Errors
    Average of Previous 10 Values
  _total: Working Set
    Average of Previous 10 Values
  WinWord: % Processor Time
    Difference between Current and Average of Last 10
  Excel: % Processor Time
    Difference between Current and Average of Last 10
```

Outlook: Handle Count
```
  Average of Previous 100 Values
```
intrude: % Processor Time
```
  Average of Previous 100 Values
```

Best #46 `(weight = highest weight / 32)`

File Write Operations/sec
```
  Difference between Averages of Prev 10 and Prev 100
```
File Write Bytes/sec
```
  Average of Previous 100 Values
```
Context Switches/sec
```
  Difference between Current and Average of Last 100
```
Number of Sections
```
  Difference between Current and Average of Last 100
```
TCP Connections Passive
```
  Difference between Averages of Prev 10 and Prev 100
```
Powerpnt: Working Set
```
  Difference between Current and Average of Last 10
```
SYSTEM: % Processor Time
```
  Average of Previous 10 Values
```
SYSTEM: Handle Count
```
  Difference between Current and Average of Last 100
```
WinZip32: Elapsed Time
```
  Difference between Current and Average of Last 10
```

Bibliography

[ANDERSON80] J. Anderson, *Computer Security Threat Monitoring and Surveillance*, J. P. Anderson Company Technical Report, Fort Washington, PA, 1980.

[DARPA99] Research and Development Initiatives Focused on Preventing, Detecting, and Responding to Insider Misuse of Critical Defense Information Systems, Workshop Report, October 1999 (http://www2.csl.sri.com/insider-misuse/).

[GOSH99] A. Ghosh, A. Schwartzbard, & M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, *USENIX Workshop on Intrusion Detection & Network Monitoring*, April 1999 (ftp://ftp.rstcorp.com/pub/papers/usenix_id99.ps).

[LANE98] T. Lane & C. Brodley, Approaches to Online Learning and Concept Drift for User Identification in Computer Security, *4th Intl. Conf. on Knowledge Discovery and Data Mining,* pp 259-263, 1998, New York (http://mow.ecn.purdue.edu/~brodley/my-papers/terran-kdd98.ps).

[LEE99] W. Lee, S.J. Stolfo, and K. Mok, A Data Mining Framework for Building Intrusion Detection Models, *Proc. IEEE Symp. on Security and Privacy*, 1999 (http://www.cs.columbia.edu/~sal/hpapers/ieee99.ps.gz).

[LITTLESTONE88] N. Littlestone, Learning Quickly When Irrelevant Attributes Abound: A New Linear-Threshold Algorithm. *Machine Learning* 2, pp. 285--318, 1988

[LUNT93] T. Lunt, A Survey of Intrusion Detection Techniques, *Computers and Security* 12:4, pp. 405-418, 1993.

[MITCHELL97] T. Mitchell, *Machine Learning*, McGraw-Hill, NY, 1997.

[MONROSE97] F. Monrose and A. Rubin, Authentication via Keystroke Dynamics, *4th Annual Conference on Computer and Communications Security* (http://avirubin.com/keystroke.ps).

[NEUMANN99] P. Neumann, *The Challenges of Insider Misuse*, SRI Computer Science Lab Technical Report, 1999 (http://www.csl.sri.com/neumann/pgn-misuse.html).

[WARRENDER99] C. Warrender, S. Forrest, B. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. *IEEE Symp. on Security and Privacy*, pp. 133-145, 1999 (ftp://ftp.cs.unm.edu/pub/forrest/oakland-with-cite.pdf).